C/C++ Contributing Editors

The New C: Integers in C99, Part 1

Randy Meyers

C has its roots in typeless languages, but it has come a long long way from its humble beginnings.

Back in the 1980s before the ANSI C Standard and function prototypes, there were occasional flame wars in the comp.lang.c newsgroup over the C type system. The wars would start when Pascal proponents would berate C for lacking strict type checking. Out of pure contrariness, the C camp would sometimes disingenuously argue against type checking in general: "C doesn't need strong typing because C programs only use three types: **int**, **char**, and pointer to **char**," wrote one poster.

This exaggeration has some humor to it because it has some truth to it as well. In much of C programming, particularly systems programming, integers of different sizes are the fundamental data type. Sure, you have arrays of them and structures of them; sometimes they represent numbers and sometimes they represent characters, but almost everything is some type of integer. The primary data type for SNOBOL is the string; for FORTRAN, the floating-point number; and for C, it is the **int**.

That is not to say that C99 does not have significant new floating-point features, because it does. However, to many C programmers, the integer is still king, and this month and the next I will cover the new integer features of C99.

No Longer the Default

C grew out of the typeless languages BCPL and B, and for a brief time was a typeless language itself [1]. It is not that those languages had no types; it is more accurate to say that they had one type, the machine word, which most operators treated as an **int**. Needless to say, declarations in those languages did not need to include a type specifier since there was only one (unnamed) type available. When Dennis Ritchie first added types to C, there were only two: **int** and **char**. I do not know if it was a nod to C's roots as a typeless language, or the popularity of **int**, or C's notable brevity, but C from its beginnings, until changed by C99, made **int** the default data type. If you declared an object or function and did not specify a type, or you implicitly declared a function by just calling it, the default type was **int**.

For example, before C99, if the following was a complete translation unit:

```
extern x;
f(y)
{
    register z = g(x) + y;
    return z;
}
```

then the variables **x**, **y**, and **z** all had type **int**, and the functions **f** and **g** had return type **int**. C99 requires that an implementation issue a diagnostic whenever a type would have defaulted to **int** under earlier definitions of C. The motivation for the diagnostic is that many uses of "implicit **int**" are errors that can be hard to spot, as in this complete translation unit:

```
int main()
{
    double d;
    d = sqrt(2.0);
    return 0;
}
```

Since **sqrt** is implicitly declared, the compiler treats it as returning an **int**. While **sqrt** will correctly store somewhere its **double** return value, **main** will not necessarily know where that return value is stored. (On some machines, integers and floating point use different registers.) After **main** loads the return value (from possibly the wrong location), it will then convert the value from **int** to **double** in order to assign it to **d**. Since the result value was already a **double** (assuming that **main** correctly found it), this results in completely scrambling the result from **sqrt**.

This sort of error can take many forms. For example, you might include the wrong header and not get a needed function declaration, or you might forget to include a type specifier when declaring an extern variable. The convenience of implicit **int** is outweighed by the difficulty of spotting the bugs it introduces. Even Ritchie himself reports he is glad to use a C compiler that forces function prototypes to be declared [2].

Note that the C99 Standard does not require that the diagnostic about implicit **int** be an error that stops the compilation. A wise implementation will make the diagnostic merely a warning, and have options to make the message an error or turn it off completely at the programmer's discretion.

long long and unsigned long long

The type **long long int** (a 64-bit or greater integer) has been an extension in some C compilers since the mid-1980s. It was added to C99 for several reasons:

- 64-bit machines are increasingly common. While the most popular mapping of C data types on such machines is **short** to 16 bits, **int** to 32 bits, and **long** to 64 bits, some 64-bit machines wish to keep **long** as 32 bits and have a new name for a 64-bit integer.
- 32-bit machines sometimes have to share data with 64-bit machines. The **long long** type gives 32-bit machines a way of handling 64-bit data.
- Floating-point programmers sometimes find it useful to have an integer that can hold all of the mantissa bits of a **double**.
- **long long** has sufficient utility and is now common enough to deserve standardization.

Note that **long long** is not part of the C++98 Standard, but it is increasingly common in C++ compilers. It is likely that a future revision of the C++ Standard will incorporate **long long** for compatibility with C.

The types **long long** and **unsigned long long** are integer data types with at least 64 bits. They may be used wherever any integer type can be used. The header **<limits.h>** now defines the macros:

- **LLONG_MIN**, the most negative number that can be stored in a **long long**
- **LLONG_MAX**, the largest positive number that can be stored in a **long long**
- **ULLONG_MAX**, the largest number that can be stored in a **unsigned long long**

Since **long long** and **unsigned long long** are at least 64 bits long, **LLONG_MAX** expands into a 19-digit number that starts with **9**. **LLONG_MIN** expands into a negative 19-digit number. **ULLONG_MAX** expands into a 20-digit number. Of course, an implementation may use more than 64 bits for **long long**, and the above limits would be adjusted up as necessary.

long long constants

The new suffix **ll** or **LL** may be added to the end of an integer constant. Some examples: **7ll**, **7LL**, **07LL**, **0x7ll**. When added to a decimal integer constant, the constant has type **long long**. When added to an octal or hexadecimal integer constant, the constant has type **long long** if the constant can be represented, or **unsigned long long** if the constant is too big for **long long**. The **ll** and **LL** suffixes can be combined with the **u** and **U** suffixes to force the constant to have type **unsigned long long**. Some examples: **7Ull**, **7LLu**, **0x7llu**, **07ULL**.

You need not use the new suffixes to have constants of type **long long** or **unsigned long long**. If an integer constant is too big to fit in any other type, it will have type **long long** or **unsigned long long** depending upon whether the constant was decimal versus octal or hexadecimal, and whether the **u** or **U** suffix was used or not. Next month's column will discuss this topic more fully, and explain how in a few very rare cases this may break old programs.

Conversions

The usual arithmetic conversions work with **long long** and **unsigned long long** as you would suspect. If you add a **long long** and a smaller integer type, the result is **long long**. If you add **unsigned long long** to a smaller integer type, the result is **unsigned long long**. If you add **long long** and **unsigned long long**, the result is **unsigned long long**. If you add **long long** or **unsigned long long** to a floating-point type, the result has the same floating-point type.

Operations

Since **long long** and **unsigned long long** are integer types, they may be used wherever any integer type can be used. This means that all of the usual arithmetic operators work on them, and they can be converted to floating-point types, etc.

printf and scanf

In C99, the **printf** and **scanf** families of functions support an optional **ll** length modifier (note that it must be in lower case). The **ll** length modifier can appear immediately before the **d**, **i**, **o**, **u**, **x**, or **X** format conversion specifier characters. For

the **printf** functions, this means that the item being printed is **long long** or **unsigned long long**. For the **scanf** functions, this means that the corresponding argument is a pointer to **long long** or **unsigned long long**. You may also use the **ll** length modifier before the **n** conversion specifier character in **printf** functions in order to store a count of the characters written thus far into a **long long** pointed to by the corresponding **printf** argument.

For example:

```
long long x;
scanf("%lld", &x);
printf("%19lld is hex %#llX\n", x, x);
```

Next Month

I have glossed over a few of the details concerning the rules giving the types of constants and the usual arithmetic conversions because they are best dealt with in next month's column. C99 permits implementations to add extra integer data types to the language, and the various rules concerning integers were generalized to handle the traditional integer types (**char**, **short**, **int**, **long**) and the new C99 integer types (**long long** and **_bool**) as well as implementation defined integers. Next month this will be covered, along with the new headers **<inttypes.h>** and **<stdint.h>**, which allow access to extended integers and which aid in increasing program portability.

Own a copy of the Standard

You can now download a copy of the C99 Standard in Adobe PDF format for $18. (In contrast, a paper copy of the Standard costs $220!). Visit **http:// www.techstreet.com/ncitsgate.html**, and enter 9899 in the first search box.

You might also want to pick up a copy of the C++ Standard for another $18. Search for standard 14882.

References

[1] Dennis Ritchie. "The Development of the C Programming Language," In Bergin and Gibson, editors, *History of Programming Languages* (Addison Wesley, 1996). Originally in *ACM SIGPLAN Notices,* Vol. 28, No. 3 (March 1993).

[2] Dennis Ritchie. "Transcript of Question and Answer Session," page 696. In Bergin and Gibson, editors, *History of Programming Languages* (Addison Wesley, 1996). (Not part of the earlier *ACM SIGPLAN Notices* publication.)

Randy Meyers *is consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*